Performance Analysis for Arm vs x86 CPUs in the Cloud

This item in japanese

Key Takeaways

- Arm-based systems and instances are readily available in public clouds such as AWS.
- The computational performance of AWS's Arm EC2 instances is similar to that of the x86_64 instances.
- Considering that Arm instances are significantly cheaper, the cost effectiveness of Arm instances are better than x86_64 instances.
- Arm instances perform better with "close to metal" applications. We hypothesize that the operating systems have received more engineering efforts to optimize for Arm than high level application frameworks.
- WebAssembly VMs outperform Docker and Node.js significantly for computational applications.

With increasing adoption of high-performance Arm-based CPUs beyond mobile devices, it is important for developers to understand Arm's performance characteristics for common server-side software stacks. In this article, we will use AWS's Arm (Graviton2) and x86_64 (Intel) EC2 instances to evaluate computational performance across different software runtimes, including Docker, Node.js, and WebAssembly. Our conclusion is that Arm is more cost effective in the cloud, especially with lightweight runtimes that are close to the underlying operating system.

Background

In a recent research paper published in Science, MIT professors Leiserson and Thompson et al. discussed one of today's most important challenges in computer engineering — the end of Moore's Law. Computer hardware, such as the CPU and GPU, had hit the quantum limit and can no longer be made much faster or smaller. That threatens 40 years of productivity and economic growth powered technology innovation. Is the technology revolution as we know it over? Yet, the paper is optimistic about our technology future.

The authors suggest that software improvements could replace Moore's Law and drive productivity growth in the years to come. To illustrate this point, they noted that re-writing machine learning algorithms from Python to C / native code could improve performance by 60,000 times!

However, we cannot just give up modern software runtimes and the developer productivities they bring, go back to pre-Java days of the 1990's, and run every application in compiled native code. Today's developers rely on high-level programming languages, tooling, and especially memory safe and portable runtimes, to deliver high-quality software products.

According to the Science paper authors, the approaches to software performance engineering are to remove software bloat and tailor software to more efficient hardware.

In this article, we will evaluate the performance gains by adopting lightweight and efficient software and hardware infrastructure in cloud computing scenarios. Specially, we run several lightweight software runtimes on both energy-efficient Arm-based CPUs (AWS Graviton2) and Intel x86 CPUs.

For the purpose of this study, we focus on single threaded performance. Most web application frameworks are running "one thread per request" by default. From the user's point of view, the web service performance is likely to be bound by how fast a single CPU can execute. This is a deliberately simple test case to illustrate the raw performance.

The benchmarks we chose are the following.

- The following two benchmarks evaluate cold start performance.
    - The nop test starts the application environment and exits.
    - The cat-sync test opens a local file, writes 128KB of text into it, and exits. It evaluates performance in making operating system calls.
- The following four benchmarks are from the Computer Languages Benchmarks Game, which provides crowd-sourced benchmark programs for over 25 programming languages. They evaluate runtime performance after starting up.
    - The nbody, repeated 50 million times, is an n-body simulation.
    - The fannkuch-redux, repeated 12 times, measures indexed access to an integer sequence.
    - The mandelbrot, repeated 15000 times, is to generate Mandelbrot set portable bitmap file.
    - The binary-trees, repeated 21 times, allocates and deallocates large numbers of binary trees.

Next, let’s look at the exact test setup and some performance numbers! The source code and scripts of all test cases are available on GitHub.

Less software bloat

To preserve software safety, security, and cross-platform portability, we run the benchmark tests in containers and VMs. One of the most popular container runtimes is Docker, which is already optimized for performance. To evaluate software stack performance, we ran the following test cases on an AWS t3.small instance, which features a physical CPU core consisting of 2 vCPUs. We idled the instance long enough to accumulate sufficient CPU credits to sustain 100% CPU bursts throughout performance tests.

Test case #1: To simulate the performance of a web application, we run the benchmark tests as a Node.js JavaScript application running inside Docker.

Test case #2: We also run the benchmark tests C/C++ native applications inside Docker with Ubuntu Server 20.04 LTS. This scenario is somewhat unrealistic since few people could compile their apps to single binary executables, and ignore the ecosystems of tools and libraries provided by runtimes like Node.js. But it serves as a comparison point for the performance we could achieve under Docker.

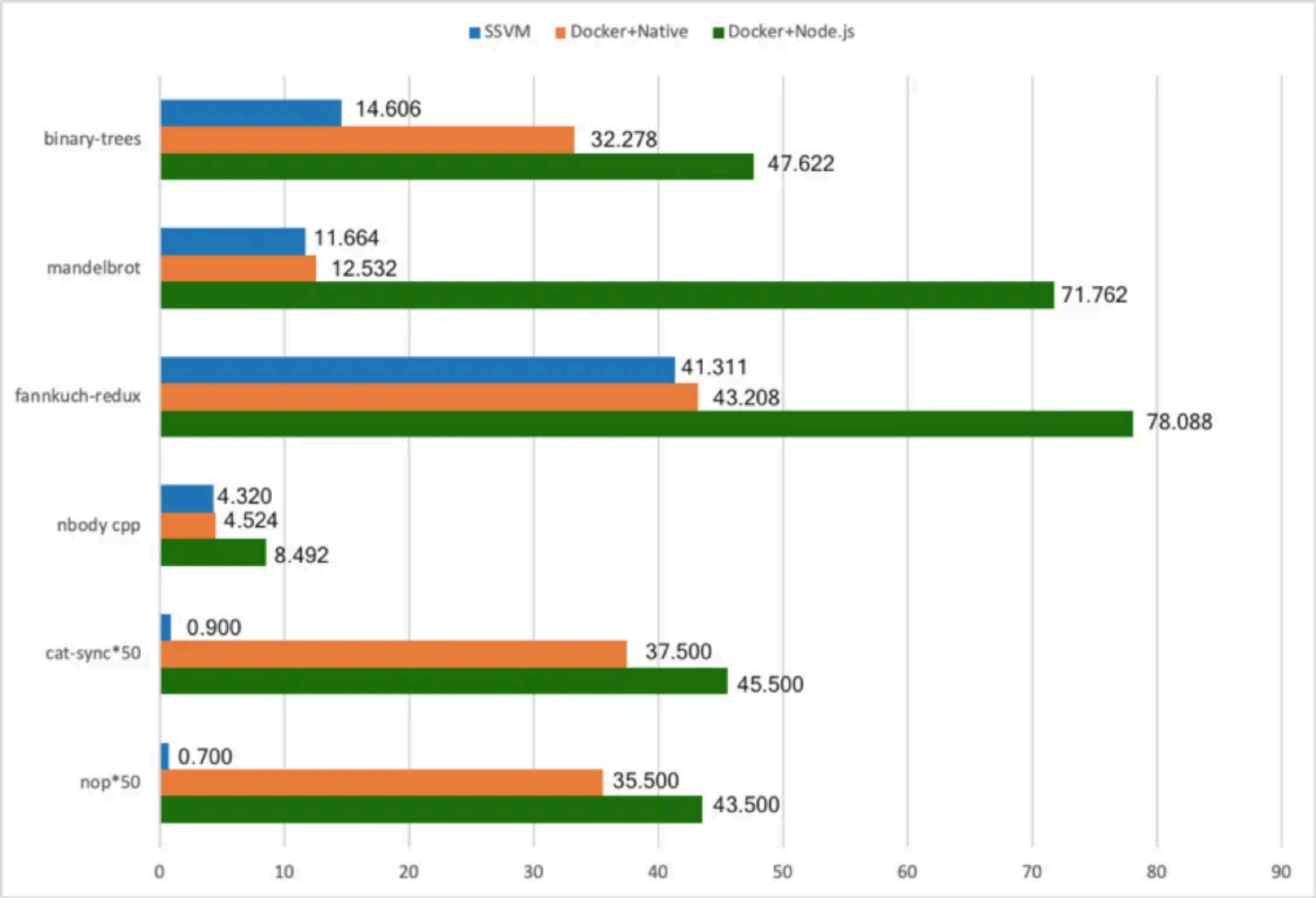
Test case #3: We run the benchmark tests in the Second State WebAssembly VM (SSVM). The programs are written in Rust and compiled to WebAssembly bytecode. The SSVM provides runtime safety, capability-based security, portability, and integration with Node.js.

Capability-based security requires the application to possess and present authorization tokens to access protected resources. In the case of SSVM, the application must explicitly declare the resources, such as file system folders, it requires access to at startup time. This design is called the WebAssembly Systems Interface (WASI).

Our software stack for the test cases is as follows.

- Amazon Linux 2 running on EC2 instances
- Docker 19.03.6-ce
- Ubuntu Server 20.04 LTS inside Docker
- Node.js v14.7.0 inside Docker
- The native executables are compiled by the LLVM 10.0.0 and Clang toolchain. On Intel architecture, we used the -O3 flag for Clang (see this section). On Graviton2, we used AWS's recommended LLVM optimization settings.
- The Second State WebAssembly VM (SSVM) 0.6.4 with AOT (Ahead-of-Time compiler) optimization

The results on Intel architecture are shown in the figure below. All numbers are for execution time in seconds. The **smaller number indicates better performance**. Note: The SSVM cold starts are orders of magnitudes faster than Docker, and hence we multiplied cold start time by 50x so that they are visible on this graph. The issue of cold starting is especially important for micro-services that are only invoked occasionally. Examples include many serverless functions that respond to occasional events, which every function call could involve a cold start of the runtime.



Excution time in seconds

Key takeaways:

- The SSVM cold starts in less than 20 milliseconds, while Docker requires 700 milliseconds or more. The SSVM is at least 30x times faster.
- For computationally intensive runtime tasks, both Docker + native and SSVM are about 2x faster than Docker + Node.js.
- Docker + native is a poor choice since it performs worse than the SSVM, while forgoing the ecosystem benefits of Node.js and JavaScript.

Programs in SSVM run even faster than native code. How is that possible? SSVM employs Ahead-of-Time (AOT) compilation techniques. It allows the compiler to optimize specifically for the machine it currently runs on, as opposed to making generic optimizations for the entire class of CPU architecture.

Switching from OS-level containers (e.g., Docker) to application-level VMs (e.g., SSVM) has resulted in significant performance gains. The SSVM does require some changes to Node.js applications, but still gives developers the benefit of runtime safety, security, portability, and the full Node.js ecosystem.

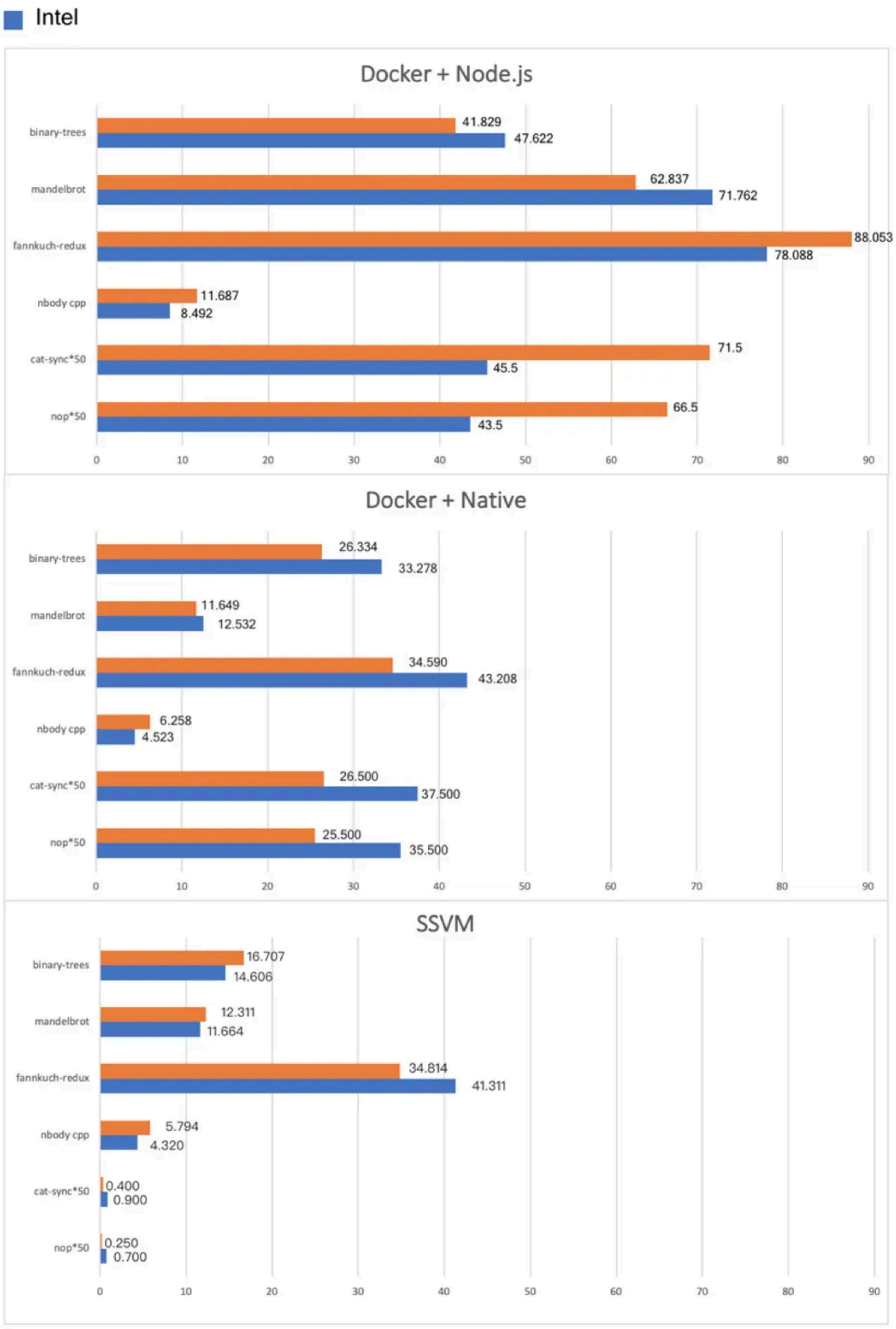
Efficient hardware

The Software Performance Engineering solution proposed by professors Leiserson and Thompson et al. is not just about removing existing software bloat. It also calls for better software utilization of hardware devices. After years of design iterations in highly constrained computing environments (e.g., on the mobile phone), the Arm architecture presents a unique opportunity to run generic software programs with high efficiency. With AWS's pioneering work in optimizing server-side virtualization for Arm, the AWS Graviton2-based Amazon EC2 instances offer the potential to improve our web application performance further. In this section, we repeated the benchmark tests on AWS t3.small (x86) and t4g.small (Arm-based Graviton2) instances. They are configured similarly, but the t4g.small (Arm) is about 24% cheaper in per hour costs.

- The t3.small instance type features Intel Xeon Platinum 8000 series processors (x86_64) with a sustained Turbo CPU clock speed of up to 3.1 GHz. The t3 instance offers 2 vCPUs running on 1 physical core and 2 GB RAM. It costs $0.0208/hr.
- The t4g.small instance type features the AWS Graviton2 CPU (Arm64 or aarch64) with a clock speed of 2.5GHz. The t4g instance offers 2 vCPUs running on 2 physical cores and 2 GB RAM. It costs $0.0168/hr.

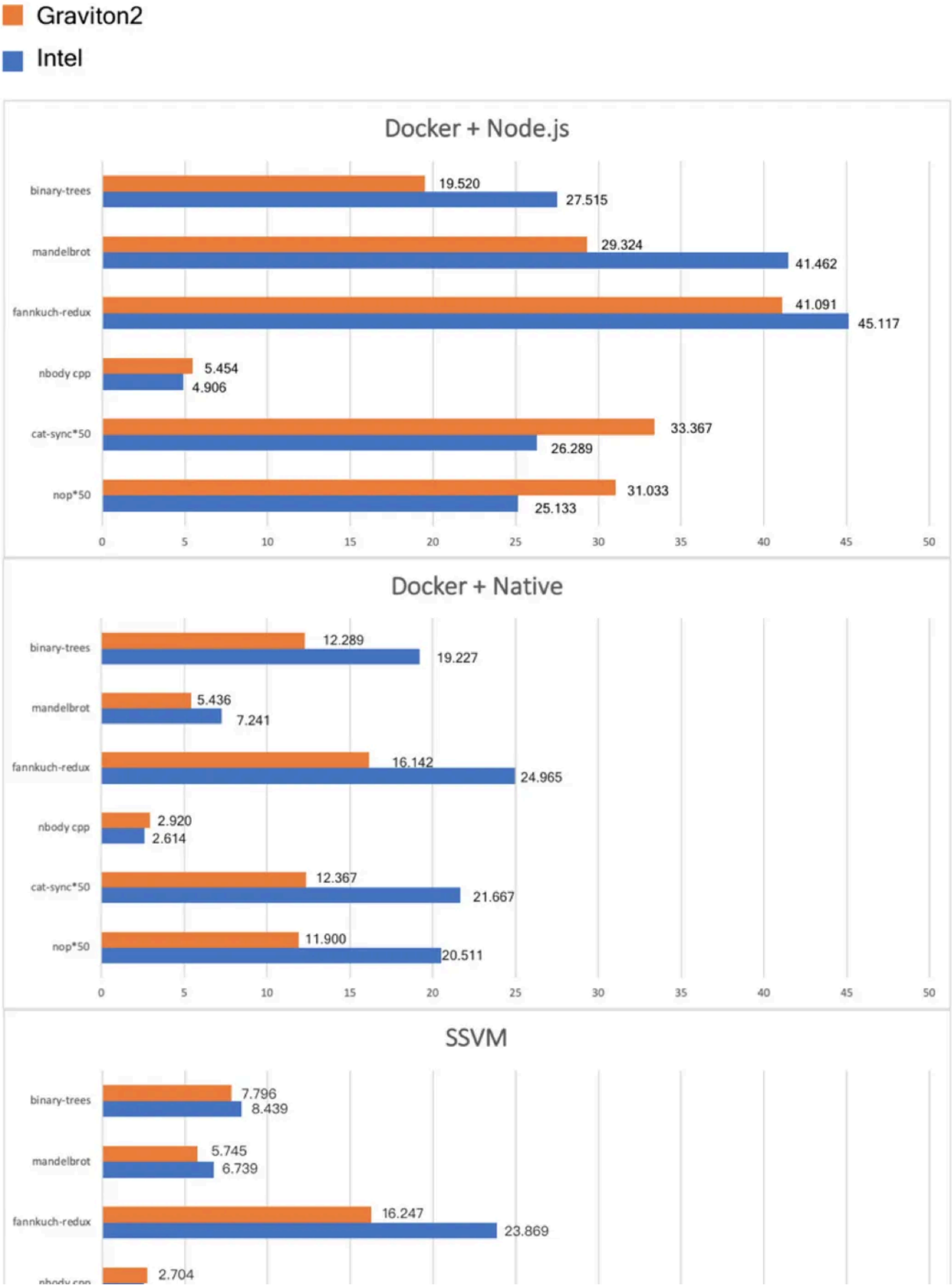
The AWS Graviton2 processor provides additional performance benefits for multi-threaded applications. However, as we discussed earlier, this article only tests single thread implementations of the benchmark algorithms.

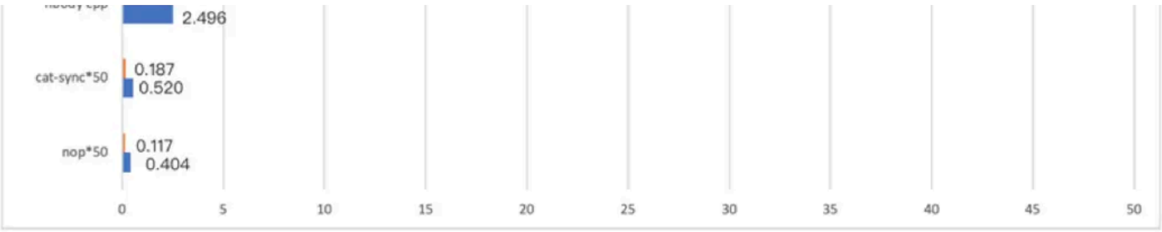
The benchmark results from Intel and Graviton2 are shown in the figure below. All numbers are for execution time in seconds. The smaller number indicates better performance.



Excution time in seconds

The following figure shows the benchmark results in terms of cost effectiveness when taking into account both of the CPU time and hourly rate for different types of CPUs in EC2. All numbers are the cost to run the benchmark operations in the unit of 0.001 cent USD. The **smaller number indicates better performance per cost.**





Cost in 0.001 cent (USD)

Key takeaways:

- The SSVM still achieves the best performance with cold start time 100x faster than Docker, and runtime performance up to 5x faster than Docker + Node.js (i.e., the mandelbrot benchmark) on both CPU platforms.
- Across the board, Graviton2 offers better cost/performance vs. Intel x86 CPUs.
- Graviton2 showed significant performance gains over Intel when running native binaries.
- The Node.js and SSVM performance comparison between Graviton2 and Intel are mixed. But considering Graviton2 instances are 24% cheaper, they come out ahead in cost/performance.

We hypothesize that the baseline Linux operating system is already optimized for ARM CPUs, allowing native binaries to take full advantage of Graviton2’s performance features. However, framework and runtime software higher in the stack, such as Node.js and SSVM, are not specifically optimized for Arm CPUs due to Arm’s relative newness in the server and cloud space. There is still significant room for improvement in Arm versions of server-side software.

What we learned

In this article, we compared commonly used algorithms and web application tasks on different compute architectures. We also compared a legacy stack Docker and Node.js vs. the new stack of SSVM (WebAssembly) and observed performance improvement of up to 100x times at cold start and up to 5x at warm runtime. There still appear to be significant room for improvements, especially with software optimization for Arm-based CPUs.

In the post-Moore era, technology could continue to lead our society's productivity growth by improving the software stack. There is plenty of room to grow at the top!

About the author

Dr. Michael Yuan is the author of five books on software engineering. His latest book, Building Blockchain Apps, was published by Addison-Wesley in Dec 2019. Dr. Yuan is the co-founder of Second State, a startup that brings WebAssembly and Rust technologies to cloud, blockchain, and AI applications. Second State enables developers to deploy fast, safe, portable, and serverless Rust functions on Node.js. Stay in touch by subscribing to the WebAssembly.Today newsletter.

Discuss